10 On Formal Analysis

In this section we show how one can use the rewrite logic definition of a language to formally analyze programs. This is achieved by using generic executability or analysis tools for rewriting logic. The formal analyses discussed in this section fall into the following categories: static type checking and type inference, type preservation and progress, concurrency analysis (e.g., dataraces and deadlocks), and model checking.

To illustrate our rewrite-based techniques for type checking, type inference, as well as for proving properties about or in connection to types, in this section and in Section 10.3 we assume the simplified version of the FUN language in Figure 10, without any control intensive features. We will define a type system for this language encoding a common typing policy. Note that here we do not consider types to be part of the language; they are nothing but an abstraction of the program, useful for static analysis purposes, to reject or warn about programs that are considered to be badly written. There are also languages in which types form an important part of the language semantics, playing a "dynamic" role in the execution of programs, that is, programs may manifest different behaviors depending upon the runtime types of some of its expressions; such a language is for example KOOL, defined in Section 7. If that is the case, then one should define types as part of the language semantics, as we did in Section 7.

10.1 Type Checking

There was little in the definition of our functional programming language so far which would forbid one from writing erroneous programs. While the domain or application specific errors are hard or impossible to catch automatically, there is a large class of errors, known as *type errors*, which often can be caught automatically with appropriate program analysis tools.

Static type checkers are special program analysis tools that take programs as inputs and report type errors, which may turn into runtime errors or misbehaviors. Type checkers are so popular nowadays and the kind of errors they catch so frequently lead to wrong behaviors, that most compilers today provide a static type checking analysis tool as a front end. The major drawback of static type checkers is that they may *reject correct programs*, so they somehow limit the possibilities of writing programs.

Typed Languages

A typed language defines a set of *types*, as well as a *typing policy* by which types can be associated to values calculated by programs.

By applying the typing policy recursively, one can mechanically associate a type to each expression in a given program. In order to do it, one needs to inspect the invocations of language constructs and check that each language construct is used properly by analyzing the relationship between its operands, their expected types and the context in which the language construct appears.

If a language construct use is found to violate the expected typing policies, then we say that a *type error* has been found. E.g., think of the expression `3 + (fun x -> x)`. While under certain special circumstances one may wish to accept such programs as correct, in general a typing policy would classify this is as a type error.

When designing a type checking tool for a typed language, one needs to take a decision on *what to do when a type error is found*.

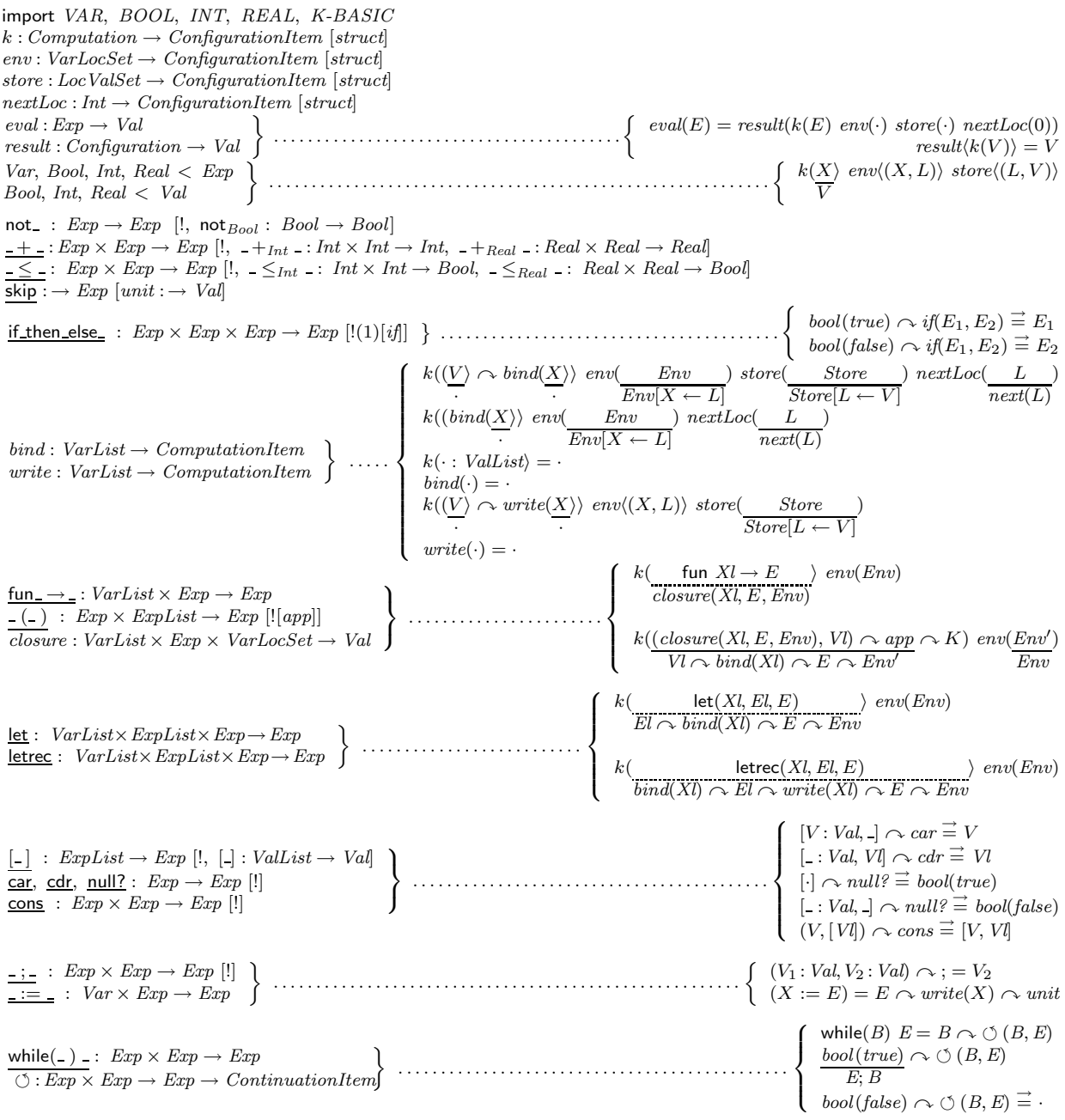

Figure 10: K definition of a simple functional language

The simplest decision is to reject the program. Better decisions are to also report where the error has been found, as well as possible causes, to users. Even better, the type checker can continue and report subsequent type errors within only one session.

One can even think of taking automatically some correcting measures, such as to convert integers to reals, or kilometers to miles, or to modify the code by introducing runtime checks, etc.

Dynamic Type Checking

At the expense of increasing their runtime overhead, some languages maintain a type together with each computed value in their store. This way, variables are associated not only values but also the types of those values. Before an operation is applied, a *dynamic type checker* validates the application of the operation. In the case of `3 + (fun x -> x)`, before the operation `_+_` is evaluated the dynamic type checker checks whether the two operands are integers; this way a runtime error will be reported.

While dynamic type checking is useful and precise, and many successful programming languages are dynamically typed, such as *Scheme*, *PERL*, *Python*, etc., many programming language designers believe that runtime checks are too expensive in practice.

Untyped Memory Models

If one simply removes the runtime checks for type consistency then one can obviously run into serious difficulties. For example, when adding an integer and a function, since they are both stored as binary numbers at some locations, without a consistency check one would just add the integer with the binary representation of (part of) a particular representation of the function, which is wrong.

Languages without runtime or dynamic type checking are said to have *untyped memory models*. Such languages essentially assume that the operations are applied on the right data, so they can increase their performance by removing the runtime type consistency checks.

Static Type Checking

To take full advantage of the increased execution speed of untyped memory models while still not sacrificing the correct runtime behavior of programs, an additional layer of certification is needed. More precisely, one needs to ensure *before execution* that all the operations will be applied on the expected type of data. This process is known under the terminology *static type checking*.

Languages admitting static type checkers, that ensure that once a program passes the type checker it will never exhibit any type error at runtime, are called *strongly statically typed*. It is quite hard or even impossible (due to undecidability arguments) in practice to devise strongly typed languages where the types are intended to state that a program is free of general purpose errors. For example, it is known to be undecidable to say whether a program will ever perform a division by zero, or if a program terminates.

To keep static type checkers decidable and tractable, one typically has to *restrict* the class of errors that types can exhibit. In the context of our simple functional programming language, like in many other languages, by a *type error* we mean one of the following:

- An application of a non-functional expression to an argument;

- An application of a function expression to arguments whose types are different from those of function's parameters;
- An assignment of an expression to an existing name, such that the type of the expression is different from the declared type of the name;
- A use of an arithmetic or boolean operation on non-integers or non-booleans, respectively;
- A use of a conditional statement where the first argument is not a boolean or where the two branches are of different type.

Besides the simplistic typing policy above, we also need to state *how the types of expressions are calculated and propagated* via *typing rules*. For example, "the type of `x + y` is `int` if the types of `x` and `y` are `int`". Similarly, "the type of a conditional is the type of its first branch if the type of its argument is `bool` and the types of its branches coincide".

In order to allow type checking in a language, one first needs to extend the language with *type declarations*, so that one can add typing information to the program. We will see later, when we discuss type inference, that in some situations types can be deduced automatically by examining how names and expressions are used.

Since declarations can occur either in a `let`/`letrec` statement or in a function (its parameter), we will slightly extend the syntax of our language to allow type declarations in addition to and at the same time with name declarations. For example, we will allow expressions like

```
let int x = 17 in x
let int x = 17 and int y = 10 in x + y
```

Besides the *basic types* `int` and `bool`, we will also introduce several other types and type constructors. Since our language is functional we will need to introduce *function types*, using the common type constructor `_->_`. Thus, we will be able to write and type-check expressions like

```
let int a = 3
in let (int -> int) p = fun(int x) -> x + a
   and int a = 5
   in a * p(2)
```

Note that, by design, we decided to first place the type of a parameter and then its name. As before, we also assume that the default calling mode is call-by-value. So expressions like the one below will also type check (note also the tricky typing of `letrec`):

```
letrec int x = 2
and    ((int,int) -> int) f = fun (int y, int z) -> y + x * z
in f(1,x)
```

As seen in Figure 10, to simplify the semantic definition of the language we assumed that all the types in a type declaration are grouped together in a list that is given as an additional argument to the corresponding language construct. It is straightforward to automatically transform the programs in this spirit (see the provided Maude code).

Defining Program Analysis Tools

The modular design of our programming language was intended not only to easily deal with changes in the design of the language, but also to facilitate the definition of *program analysis tools*.

In fact, the definition of a program analysis tool can be seen as very similar in spirit to defining a semantics to a programming language. The meaning a program analysis tool has for a program may, of course, be quite different from that of the semantics of the programming language, because the two may look at two quite different aspects of a program.

However, we will again take advantage of the efficient executable environment provided by Maude through its fast implementation of rewriting, and this way obtain not only executable but also quite efficient program analysis *tools for free*, from just their mathematical rigorous definition.

A Big-Step Definition of a Type Checker

As we saw at the begining of the class, big-step semantics are simpler and more efficiently executable than other semantics when they work. There were two problems making big-step semantics inappropriate: concurrency and control-intensive statements. Since we have none of these problems when we define a type-checker for our language, a big-step semantics is appropriate in this case.

Adding Types to the Syntax

There are very few changes that need to be done to the syntax of our functional programming language in order to allow type declarations. First, of course, one has to introduce the types:

```
fmod TYPE is
  sorts Type TypeList .
  subsort Type < TypeList .
  op nil : -> TypeList .
  op _,_ : TypeList TypeList -> TypeList [assoc id: nil] .
  ops int bool none : -> Type .
  op _->_ : TypeList Type -> Type [gather(e E)] .
  op list_ : Type -> Type [prec 0] .
endfm
```

Besides `bool` and `int`, note that there is one more basic type, `none`, which will be the type of expressions that are supposed to evaluate to `unit` (assignments, loops, etc.). We also define lists of types, for two reasons: (1) the type of the argument of a function is a list of types (an alternative would have been to introduce product types for the same purpose; in fact, one can think of lists of types as product types), and (2) as usual, we prefer to process expressions in bulk, so typing a list of expressions results in a list of types.

The type of a function without parameters returning an integer will be therefore `nil -> int`. To avoid writing parentheses, note that the function type constructor, `_->_` has been defined as right-associative. Note that this is opposite to the definition of function application, which was defined left-associative, and that these two conventions are consistent.

The language constructs that declare variables, i.e., functions, let and letrec, need to be changed to assign a type to each declared variable. To simplify our semantics later, we prefer to declare the types of all the corresponding variables in a list:

```
  op fun__->_ : TypeList NameList Exp -> Exp .
...
  op let : TypeList NameList ExpList Exp -> Exp .
...
  op letrec : TypeList NameList ExpList Exp -> Exp .
```

No other changes are needed to the syntax of the language. As already mentioned, one can easily transform the code to offer the user a nicer language interface, where each variable is prefixed with its type (see the provided file `fun-type-checking-semantics.maude`, which is also explained below).

Defining a Static Type Checker

The general idea underlying a static type checker is to recursively analyze each language construct occurring in a given program, to chack that its operands satisfy the type constraints, and then to assign a type to the expression created by that new construct.

The type of a name cannot be changed by variable assignment constructs, so *a name has its declared type in all its occurrences within its scope*; but clearly one can differentiate between static and dynamic scoping. We will only consider static scoping.

Exercise 1 (Very hard!). *Define a static type checker for the dynamically scoped version of `FUN`. Can the task be accomplished precisely? If not, choose some acceptable trade-offs so that the type checker stays reasonably efficient.*

Defining the State Infrastructure

Any software analysis tool has a configuration or state that it maintains as it traverses the program to be analyzed. This is also what happened in the definition of the semantics of the language. In fact, there is a striking similarity in giving various executable semantics to a given syntax: a programming language semantics is just one possibility; a type checker is another one; other software analysis tools can be defined similarly, including complex type systems or abstractions.

A type checker ignores the concrete values bound to names, but it needs instead to *bind names to their types*. We can realize that by defining a special state attribute, called `TypeEnv`, which contains a mapping from names to their current types. In order to do define this module, we need to first define lists of names and of types. The former were already defined in the semantics of `FUN`, and the latter are similar, so we do not discuss them here. See the provided file `fun-type-checking-semantics.maude` for the complete Maude specification of the static type checker discussed here.

The following module can simply "cut-and-paste" the module `ENVIRONMENT` defined by the executable semantics of `FUN`; however, one needs to replace locations by types. One can also use Maude's parameterized modules, but we do not do it here.

```
fmod TYPE-ENVIRONMENT is
  protecting NAME-LIST .
  protecting TYPE .
  sort TypeEnv .
  op empty : -> TypeEnv .
  op [_,_] : Name Type -> TypeEnv .
```

```
  op __ : TypeEnv TypeEnv -> TypeEnv [assoc comm id: empty] .
  op _[_] : TypeEnv Name -> [Type] .
  op _[_<-_] : TypeEnv NameList TypeList -> TypeEnv .
  var X : Name .  vars TEnv : TypeEnv .  vars T T' : Type .
  var Xl : NameList .  var Tl : TypeList .
  eq ([X,T] TEnv)[X] = T .
  eq TEnv[nil <- nil] = TEnv .
  eq ([X,T] TEnv)[X,Xl <- T',Tl] = ([X,T'] TEnv)[Xl <- Tl] .
  eq TEnv[X,Xl <- T,Tl] = (TEnv [X,T])[Xl <- Tl] [owise] .
endfm
```

In this case, no other information but the type environment is needed by the static type checker. However, to preserve our general semantic definition methodology, we prefer to define a state of the type checker which contains only one state attribute for the time being, the type environment. Extensions of the language may require new state attributes in the future:

```
fmod TYPE-STATE is
  extending TYPE-ENVIRONMENT .
  sorts TypeStateAttribute TypeState .
  subsort TypeStateAttribute < TypeState .
  op empty : -> TypeState .
  op __ : TypeState TypeState -> TypeState [assoc comm id: empty] .
  op tenv : TypeEnv -> TypeStateAttribute .
endfm
```

We can now finalize the state infrastructure by defining the following module, which, like for the semantics of the language, provides an abstract state interface for the subsequent modules:

```
fmod HELPING-OPERATIONS is
  protecting GENERIC-EXP-LIST-SYNTAX .
  protecting TYPE-STATE .
  op initialTypeState : -> TypeState .
  op _[_] : TypeState Name -> [Type] .
  op _[_<-_] : TypeState NameList TypeList -> TypeState .
  var S : TypeState .  var TEnv : TypeEnv .
  var X : Name .  var Xl : NameList .  var Tl : TypeList .
  eq initialTypeState = tenv(empty) .
  eq (tenv(TEnv) S)[X] = TEnv[X] .
  eq (tenv(TEnv) S)[Xl <- Tl] = tenv(TEnv[Xl <- Tl]) S .
endfm
```

`S[X]` gives the type associated to a name `X` in a state `S`, while `S[Xl <- Tl]` updates the types of the names in the list `Xl` to `Tl`.

Typing Generic Expressions

Like in the case of the semantic definition of `FUN`, we need to "evaluate" an expression to a "value". However, this time, the values that expressions evaluated to are types. Unlike in the definition of the semantics of the language where side effects could modify the values bound to names, *the type bound to a name cannot be modified.*

Therefore, we only need one operation, say `type`, returning the type of an expression; there is no need to propagate states and "side effects". Since we eventually need to handle lists of names, expressions and types, because of `let` and `let rec`, we prefer to define it directly on lists of expressions. Note that the result of this operation is a bracketed sort, or a kind in Maude terminology, reflecting the fact that some expressions may now be typed.

```
fmod GENERIC-EXP-STATIC-TYPING is
  extending HELPING-OPERATIONS .
  vars E E' : Exp .  var El : ExpList .  var S : TypeState .  var I : Int .  var X : Name .
  op type : ExpList TypeState -> [TypeList] .
  eq type((E,E',El), S) = type(E, S), type((E',El), S) .
  eq type(I, S) = int .
  eq type(X, S) = S[X] .
endfm
```

Typing Arithmetic and Boolean Expressions

To keep the semantics of the type checker compact, we will use conditional equations in what follows.

We can now define the *typing rules* for the arithmetic and boolean operators simply as their corresponding conditional equations. For example, the type of `E + E'` in a state `S` is `int` if both `E` and `E'` have the type `int` in `S`:

```
...
 ceq type(E + E', S) = int if (int,int) := type((E,E'), S) .
...
```

The typing of boolean expressions can be defined similarly:

```
...
 ceq type(E geq E', S) = bool if (int,int) := type((E,E'), S) .
 ceq type(E and E', S) = bool if (bool,bool) := type((E,E'), S) .
 ceq type(not E, S) = bool if bool := type(E, S) .
...
```

Note that being able to type a list of expressions to a list of types, as opposed to typing just one expressions, helps us write more compact and readable definitions.

Typing the Conditional

The typing policy of `if_then_else_` states that its first argument should be of type `bool`, while the other two arguments should have the same type. This policy can be simply stated as follows:

```
...
 ceq type(if BE then E else E', S) = T
    if (bool,T,T) := type((BE,E,E'), S) .
...
```

Note that `T` occurs twice in the pattern in the condition. This is perfectly correct; remember that the same matching algorithm used for the left-hand-sides of equations is used once the matched term is reduced to its normal form.

Typing Functions

Functions and their applications are, in some sense, the trickiest to type. Function declarations can be typed as follows:

```
eq type(fun Tl Xl -> E, S) = Tl -> type(E, S[Xl <- Tl]) .
```

So the body is typed in the declaration environment updated with a binding of the function's parameter, if any, to its declared type. The type of an empty list of arguments is `nil`.

The typing policy of a function invocation `E El` is as follows: `E` should first type to a function type, say `Tl -> T`; then `El` should type to the same type `Tl` that occurred in the type of `E`; if this is the case, then the type of the function application is, as expected, `T`. All these words can be replaced by a straightforward conditional equation (note, again, the double occurrence of a variable in the pattern):

```
ceq type(E El, S) = T if ((Tl -> T), Tl) := type((E,El), S) .
```

Note that our type system currently does not allow parametric polymorphism (i.e., the capability of expressions to have parametric types), which is a serious limitation of a statically typed higher-order language. We will address polymorphism shortly.

Typing `let`

The type of a `let` expression is the type of its body in the state obtained after binding its names to the types of the corresponding expressions:

```
ceq type(let (Tl,Xl,El,E), S) = type(E, S[Xl <- Tl]) if Tl := type(El, S) .
```

The type of a `let` remains undefined if its bindings cannot be properly typed. This is assured by the fact that `Tl` is declared as a variable of sort `TypeList`, so `type(El, S)` must reduce to a proper list of types in order for the equation to be applied.

Typing `let rec`

Typing of `let rec` is relatively easy once we have the current infrastructure. In order to find its type, one

- first blindly binds all its names to their declared types;
- then checks the type consistency of the bound expressions to their corresponding names' types;
- then, if each of those are correctly typed, returns the type of its body expression in the new state:

```
ceq type(letrec (Tl,Xl,El,E), S) = type(E, S[Xl <- Tl]) if Tl := type(El, S[Xl <- Tl]) .
```

Typing Lists

The typing policy of lists is that each element in the list should have the same type, say `T`; if this is the case, then the type of the list expression is `list T`. For example, `[1 :: 2 :: 3]` is typed to `list int`, while `[1 :: true :: 3]` generates a type error.

How about the type of the empty list, `[nil]`? It must be a list of some type, but what type? This is a subtle problem, related, again, to polymorphism. We will discuss polymorphism shortly. For now, to keep our language type-checkable, we make the strong and admittedly unsatisfactory assumption that all empty lists have type `list int`. Hence, we need an operator that takes an `ExpList` and returns the type of its elements. With that, the type of a list can be simply defined as follows:

```
 op collapseType : TypeList -> Type .
 eq collapseType(nil) = int .
 eq collapseType(T,T,Tl) = collapseType(T,Tl) .
ceq type([El], S) = list T if T := collapseType(type(El, S)) .
```

The typing rules of the list operations are self-explanatory:

```
ceq type(car(E), S) = T if list T := type(E, S) .
ceq type(cdr(E), S) = list T if list T := type(E, S) .
ceq type(cons(E,E'), S) = list T if (T, list T) := type((E,E'), S) .
ceq type(null?(E), S) = bool if list T := type(E, S) .
```

Typing Assignment and Sequential Composition

A variable assignment statement should not modify the type of the name on which it applies its side effect. Also, since we decided to evaluate assignment statements to the value `nothing`, we also type them to the special type `none`, so the type system will signal if one uses them in wrong contexts, for example as integers:

```
ceq type(X := E, S) = none if type(X, S) = type(E, S) .
```

The type of `E ; E'` is the type of `E'`, but only if `E` also can be correctly typed, say to some type `T`:

```
ceq type(E ; E', S) = T' if (T, T') := type((E,E'), S) .
```

Typing Loops

Loops type check as follows:

```
ceq type(while Cond Body, S) = none
   if (bool,T) := type((Cond, Body), S) .
```

Note that the type of a loop is `none` only if its condition types to `bool` and its body to some proper type. We could have just as well required that the type of its body be `none`, or that the type of the loop is the type of its body, etc.

Putting the Type Checker Together

Like in the semantics of FUN, we can now put everything together by defining one unifying module importing all the various features. This module also defines an operation that can check and calculate the type of an expression in the initial state:

```
  op type : Exp -> [Type] .
  eq type(E) = type(E, initialTypeState) .
```

For writing programs in a more human readable way, we can also define now some sytactic sugar notations:

```
--- some syntactic sugar
---   for bindings
  sorts Binding Bindings .
  subsort Binding < Bindings .
  op __=_ : Type Name Exp -> Binding .
  op (_,_,_) : TypeList NameList ExpList -> Binding .
  op nil : -> Bindings .
  op _and_ : Bindings Bindings -> Bindings [assoc id: nil prec 100] .
  ops (let_in_) (letrec_in_) : Bindings Exp -> Exp .
  var T : Type .  var X : Name .  vars Tl Tl' : TypeList .
  vars Xl Xl' : NameList .  vars El El' : ExpList .
  eq (T X = E) = (T,X,E) .
  eq (Tl,Xl,El) and (Tl',Xl',El') = ((Tl,Tl'), (Xl,Xl'), (El,El')) .
  eq    let (Tl,Xl,El) in E =    let(Tl,Xl,El,E) .
  eq letrec (Tl,Xl,El) in E = letrec(Tl,Xl,El,E) .
---   for functions
  sorts Parameter ParameterList .
  subsort Parameter < ParameterList .
  op __ : Type Name -> Parameter [prec 0] .
  op `(`) : -> ParameterList .
  op _,_ : ParameterList ParameterList -> ParameterList [assoc id: ()] .
  op (_,_) : TypeList NameList -> ParameterList .
  op fun_->_ : ParameterList Exp -> Exp .
  eq () = (nil,nil) .
  eq T X = (T,X) .
  eq (Tl,Xl),(Tl',Xl') = ((Tl,Tl'),(Xl,Xl')) .
  eq fun (Tl,Xl) -> E = fun__->_(Tl,Xl,E) .
```

Polymorphism

Let us consider a "projection" function `fun (x,y) -> x` which takes two arguments and always returns its first argument. In the context of our typed language above where the user is expected to assign a type to each declared variable, one is therefore expected to write it as `fun (Tx x,Ty y) -> x`, for some concrete types `Tx` and `Ty`. Unfortunately, that would imply that this function can only be used in contexts where arguments of types `Tx` and `Ty` are passed to it; this is very inconvenient, because one would like to have a generic projection function, that works on arguments of any type. In other words, we would like a *polymorphic* projection function. *Polymorphism* is the capability of a fragment of code to be used in different contexts, where codes of different types are expected.

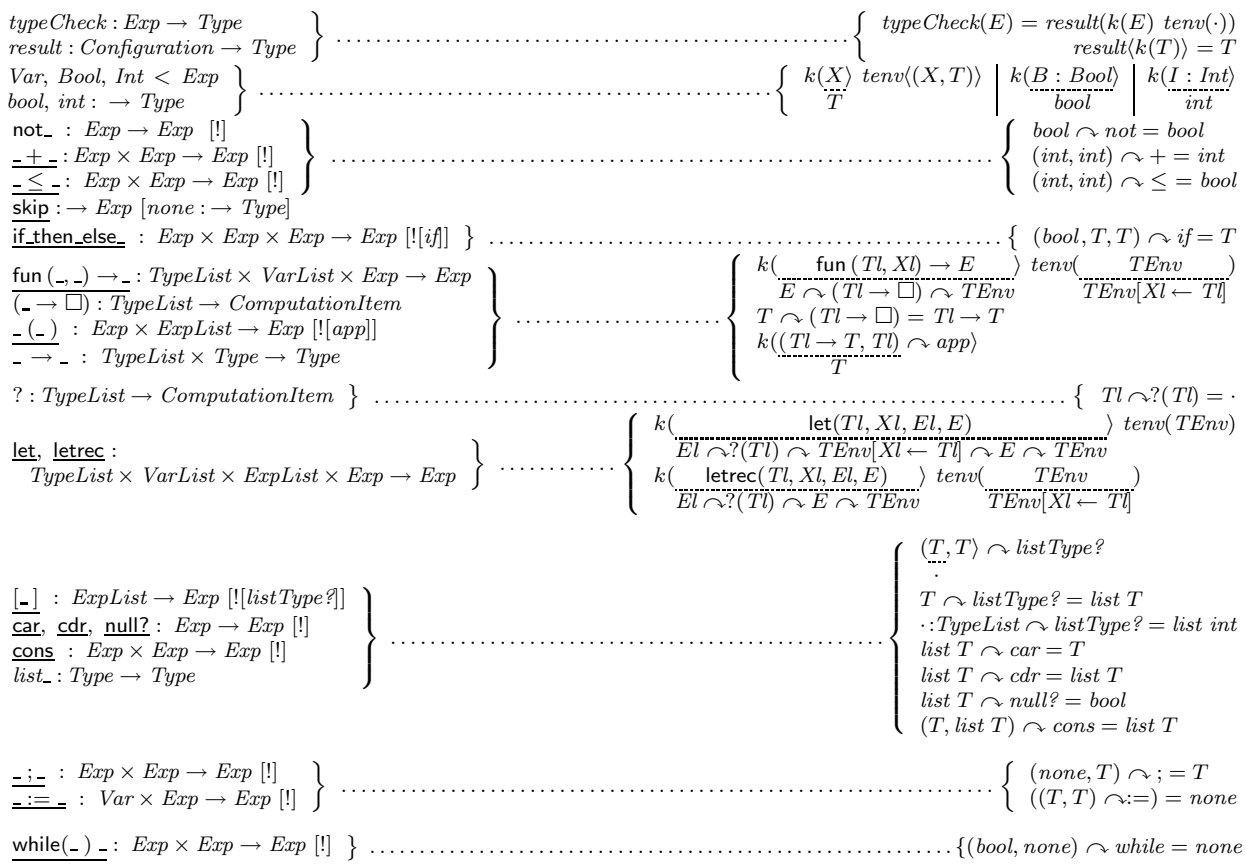

Figure 11: K definition of a type checker for the simple functional language

Exercise 2 *Add polymorphism to the type system above.*

Hint. To add generic types to the type system, we need *type variables*. Type variables can be added by declaring `Qid` (quoted identifiers are defined in the Maude builtin module `QID`) to be a subsort of `Type`. Then one can write functions like the projection one above as follows: `fun ('a x,'b y) -> x`. If we specifically want the function to take arguments of the same type, but still be polymorphic in that type, then we write it: `fun ('a x,'a y) -> x`. When a function is applied, one should check that the passed arguments have types consistent with those declared for the function's parameters. For example, the first projection function above can be safely called on arguments of types "`(int, int-> int)`", but the second cannot; the second can onlu be called on arguments of the same type, for example "`(int -> int, int-> int)`". One will also need a way to generate fresh type variables, to be able to assign a polymorphic list type to empty lists.

Modifying the K Language Definition Into a Type Checker

Figure 11 shows a relatatively straightforward translation of the K language definition from Figure 10 into a K-definition of a type checker.

Exercise 3 *Translate the K-definition of the type checker in Figure 11 into Maude.*

10.2 Type Inference

Checking that operations are applied on arguments of correct types statically has, as we have already discussed, *two major benefits*:

- Allows efficient implementations of programming languages by assuming untyped memory models and therefore removing the need for runtime consistency checks, and
- Gives rapid feedback to users, so they can correct errors at early stages.

These advantages are typically considered so major in the programming language community, that the *potential drawbacks* of static typing, namely

- Limiting the way programs are written by rejecting programs which do not type-check, and
- Adding typing information may be space and time consuming,

are often ignored.

The first drawback is usually addressed by rewriting the code in a way that passes the type checking procedure. For example, programs passing a function as an argument to itself, which do not type check because of the recursive nature of the type of that function, can be replaced by equivalent programs using `letrec`.

It is, of course, desirable to require the user to provide *as little type information as possible* as part of a program. In general, depending on the application and domain of interest, there are quite *subtle trade-offs* between the amount of user-provided annotations and the degree of automation of the static analyzer.

In the previous section we discussed a simple static type checker which requires the user to provide a type for each declared name. Since some names can be functions taking functions as arguments, the amount and shape of these type annotations can be quite heavy. In some cases they may even exceed the size of the program itself. In this section we address the problem of reducing the amount of required type information by devising automatic procedures that *infer the intended types from how the names are used*.

How much type information can be automatically and efficiently inferred depends again upon the particular domain of interest and its associated type system. In this section we will see an extreme fortunate situation, in which *all* the needed type information can be inferred automatically.

More precisely, we will discuss a classical procedure that *infers all the types* of all the declared names in our functional programming language. A similar technique is implemented as part of the ML and OCaml languages. By "type", we here mean the language specific types, that is, those that were explicitly provided by users to the static type checker that we discussed.

Type Inference

Programs written in the functional language discussed so far contain all the information that one needs in order to infer the intended type of each name. By carefully collecting and using all this information, one can type check programs without the need for the user to provide any auxiliary type information.

Suppose that one writes the expression `x + y` as part of a larger expression. Then one can infer from here that `x` and `y` are both intended to be integers, because the arithmetic operation `_+_` is defined only on integers! Similarly, if

```
if x then y else z
```

occurs in an expression, then one can deduce, thanks to the typing policy associated to conditionals, that `x` has type `bool` and that `y` and `z` have the same type. Moreover, from

```
if x then y else z + t
```

one can deduce that `z` and `t` are both of type `integer`, implying that `y` is also `integer`. Type inference and type checking work smoothly together, in the sense that one implicitly checks the types while inferring information. For example, if

```
if x then y else z + x
```

is seen then one knows that there is a typing error because the type of `x` *cannot* be both `integer` and `bool`.

The type of functions can also be deduced from the way the functions are used. For example, if one uses `f(x,y)` in some program then one can infer that `f` takes two arguments. Moreover, if `f(x,x)` is seen then one can additionally infer that `f`'s arguments have the same type.

There can be possible that the result type of a function as well as the types of its arguments can all be inferred from the context, without even analyzing the definition of the function. For example, if one writes

```
f(x,y) + x + let x = y in x
```

then one first infers that `f` must return an `int` because it is used as an argument of `_+_`, then that `x` is an integer for the same reason, so the type of `f`'s first argument is `int`, and finally, because of the `let` which is used in a context of an `int`, that the type of `y` is also `int`. Therefore, the type of `f` must be `(int,int) -> int`. In fact, the types of all the names occurring in the expression above were deduced by just analyzing carefully how they are used.

Let us now consider an entire expression which evaluates properly, for example one defining and using a factorial function:

```
letrec f(n) = if n eq 0
              then 1
              else n * f(n - 1)
in f(5)
```

How can one automatically infer that this expression is type safe? Since `eq` takes two `int`s and returns a `bool`, one can deduce that the parameter of the function is `int`. Further, since the `f` occurs in the context of an integer and takes an expression which is supposed to be an integer as argument, `n - 1`, one can deduce that the type of that `f` is `int -> int`. Because of the semantics of `letrec`, the bound `f` will have the same type, so the type of the entire expression will be `int`.

One can infer/check the types differently, obtaining the same result.

Polymorphic Functions

Let us consider again the polymorphic "projection" function which takes two arguments and returns its first one: `fun(x,y) -> x`. Without any additional type information, the best one can say about the type of this expression is that it is (t_1, t_2) `->` t_1, for some types t_1 and t_2. This function can be now used in any context in which its result type equals that of its first argument, such as, `((int`

`-> int), int) -> (int -> int)`. The type of such polymorphic functions should be thought of as the *most general*, in the sense of the *least constrained*, type that one can associate them.

Let us consider several other polymorphic functions. For example, the type of the function `fun (x,y) -> x y`, which applies its first argument, expected therefore to be a function, to its second argument, is $(t_1$ `->` t_2`,` $t_1)$ `->` t_2.

How can one infer the most general type of an expression then, by just analyzing it syntactically? The process of doing this is called *type inference* and consists of two steps:

1. Collect information under the form of type parametric constraints by recursively analyzing the expression;

2. Solve those constraints.

Collecting Type Information

In order to collect type information from an expression, we traverse the expression recursively, assign generic types to certain names and expressions, and then constrain those types. Let us consider, for example., the expression

```
fun (x,y) -> if x eq x + 1 then x else y
```

and let us manually simulate the type information collecting algorithm.

We have a function expression. Without additional information, the best we can do is to assume some generic types for its parameters and then calculate the type of its body. Let t_x and t_y be the generic types of `x` and `y`, respectively. If t_e is the type of function's body expression, then the type of the function will be

(t_x, t_y) `->` t_e.

Let us now calculate the type t_e while gathering type information as we traverse the body of the function.

The body of the function is a conditional, so we can now state a first series of constraints by analyzing the typing policy of the conditional: its condition is of type `bool` and its two branching expressions must have the same type, which will replace t_e. Assuming that t_b, t_1 and t_2 are the types of its condition and branching expressions, respectively, then we can state the type constrains

t_b `=` `bool`, and
t_1 `=` t_2.

We next calculate the types t_b, t_1 and t_2, collecting also the corresponding type information. The types t_1 and t_2 are easy to calculate because they can be simply extracted from the type environment: t_x and t_y, respectively (thus, the type equation t_1 `=` t_2 above is in fact t_x `=` t_y).

To type the condition `x eq x + 1`, one needs to use the typing policy of `_eq_`: takes two arguments of type `int` and returns type `bool`. Thus t_b is the type `bool`, but two new type constraints are generated, namely

t_3 `=` `int`, and
t_4 `=` `int`,

where t_3 and t_4 are the types of the two subexpressions of `_eq_`. t_3 evaluates to t_x; in order to evaluate t_4 one needs to apply the typing policy of `_+_`: takes two `int` arguments and returns an `int`. These generate the constraint

t_x `=` `int`.

Therefore, after "walking" through all the expression and analyzing how names were used, we collected the following useful typing information:

t_x `=` t_y,
t_x `=` `int`, and
the type of the original expression is (t_x, t_y) `->` t_x.

After we learn how to solve such type constraint equational systems we will be able to infer from here that the expression is correctly typed and its type is `(int, int) -> int`.

Let us now consider the polymorphic function

```
fun (x,y) -> (x y) + 1.
```

After assigning generic types t_x and t_y to its parameters, while applying the typing policy on `_+_` one can infer that the type of the expression `(x y)`, say $t_{(xy)}$, must be `int`. Also, the result type of the function must be `int`. When calculating $t_{(xy)}$, by applying the typing policy for function application, which says that the type of `(E El)` for expression `E` and expression list `El` is `T` whenever the type of `E` is `Tl -> T` and that of `El` is `Tl`, one infers the type equational constraint t_x `=` t_y `->` $t_{(xy)}$. We have accumulated the following type information:

$t_{(xy)}$ `=` `int`,
t_x `=` t_y `->` $t_{(xy)}$, and
the type of the function is (t_x, t_y) `->` `int`.

We will be able to infer from here that the type of the function is `(`t `->` `int,` t`)` `->` `int`, so the function is *polymorphic in t*.

The Constraint Collecting Technique

One can collect type constraints in different ways. We will do it by recursively traversing the expression to type only once, storing a type environment as well as type constrains. If the expression is

- a *name* then we return its type from the type environment and collect no auxiliary type constraint;
- an integer or a bool constant, then we return its type and collect no constraint;
- an *arithmetic operator* then we recursively calculate the types of its operand expressions accumulating the corresponding constraints, then add the type constraints that their type is `int`, and then return `int`;
- a *boolean operator* then we act like before, but return `bool`;

- a *conditional* then calculate the types of its three subexpressions accumulating all the type constraints, then add two more constraints: one stating that the type of its first subexpression is `bool`, and another stating that the types of its other two subexpressions are equal.
- a *function* then generate a fresh generic type for its parameter, if any, bind it in the type environment accordingly, calculate the type of the body expression in the new environment accumulating all the constraints, and then return the corresponding function type;
- a *function application* then generate a fresh type t, calculate the type of the function expression, say t_f, and that of the argument expression list, say t_l accumulating all the type constraints, and then add the type constraint t_f `=` t_l `->` t.
- a *declaration* of names in a `let`, then calculate the types of the bound expressions, bind them to the corresponding names in the type environment, and then calculate and return the type of `let`'s body; there are no additional type constrains to accumulate (besides those in the bound expressions and body);
- a *declaration* of names in a `letrec`, then assign some generic types to these names in the type environment, calculate the types of bound expressions and accumulate the constraint(s) that these must be equal to the generic types assigned to `letrec`'s names, and finally calculate and return the type of `letrec`'s body;
- a *list expression*, then calculate the types of all the elements in the list accumulating all the corresponding constraints, and then add the additional constraints that all these types must be equal (let `T` be that type) and finally return the type of the list expression as `list T`; if the list is empty, then `T` must be a fresh type variable;
- a *list operation*, such as `car`, `cdr`, `cons` or `null?`, then calculate the types of the corresponding subexpressions accumulating the constraints and then adding new, straightforward type constraints; for example, in the case of `cons(E,E')`, add the constraint that the type of `E'` must be `list T`, where `T` is the type of `E`;
- an *assignment* `X := E`, then calculate the types of `X` and `E`, and then add the constraint that these two types must be equal.
- a *sequential composition* `E ; E'`, then calculate the types of `E` and `E'` accumulating all the type constraints, then return the type of `E'` as the type of the composition;
- a *loop* `while Cond Body`, then calculate the types of `Cond` and `Body`, add the constraint that the type of `Cond` must be `bool`, and then return the type `none`.

Exercise 4 *The type inference technique discussed below is based on a divide-and-conquer of the main task: we first collect all the type constraints as equations, and then solve the type constraints. Define a type inferencer that solves the equational type constraints on-the-fly, as they are generated; this would correspoond to doing "online unification".*

Defining a Type Constraint Collector

We will use the same programming language and analysis tool equational definitional technique that we used to give big-step semantics to `FUN` and to define its static type checker. Since we will not need type declarations (all types will be automatically inferred, we can just import the syntax of the language unchanged from the executable semantics definition.

The `TYPE-STATE` (module) of our type constraint collector will need to contain three attributes: a *type environment*, a *set of type equations*, and a *counter* needed to generate fresh types.

The constraint collector is defined in the same style as the other tools defined so far, namely inductively over the structure of the syntax. More precisely, we will define an operation

```
preType :  Exp TypeState -> [Pair]
```

which calculates the generic type of an expression in a given state, as well as a new state containing all the accumulated constraints. Since experience showed us that it is more convenient to handle lists of expressions in block, we will actually define the `preType` operator directly on lists of expressions.

We could have defined two distinct operations instead, one returning the type and the other the constraints (which can be regarded as "side effects"). However, `preType` will combine both those operators into only one, thus allowing us to design a more compact and more easily understandable specification.

We call the type returned by `preType` a *pre-type*, because in order to calculate the final type we need to solve the equational constraints. We will first focus on defining `preType` and then on solving the type constraints.

In what follows, we only discuss the interesting aspects of the type constraint collector. More details can be found in the file `fun-type-inference-semantics.maude` on the website, which you are supposed to complete as part of your homework.

Defining Types

Like for the type checker, we introduce a specification defining types. Because of the more complex nature of type inference, and especially because of its ability to infer types in the context of polymorphism, we will need more structure on types than before. We declare a subsort `TypeVar` of the `Type` and a *type variable constructor* "`op t : Nat -> TypeVar`", which will be used to generate fresh types as well as to represent polymorphic types:

```
fmod TYPE is protecting NAT .
  sorts Type TypeVar TypeList .
  subsort TypeVar < Type < TypeList .
  op nil : -> TypeList .
  op _,_ : TypeList TypeList -> TypeList [assoc id: nil] .
  ops int bool none : -> Type .
  op _->_ : TypeList Type -> Type [gather(e E)] .
  op list_ : Type -> Type [prec 0] .
  op t : Nat -> TypeVar .
endfm
```

Defining Type Equations

A *type equation* is a commutative pair of types. To handle blocks of constraints, we allow equations of lists of types (defined the usual way in a module `TYPE-LIST`) and translate them into sets of

corresponding equations. The set of equations is kept small by removing the useless ones:

```
fmod EQUATIONS is
  protecting TYPE .
  sorts Equation Equations .
  subsorts Equation < Equations .
  op empty : -> Equations .
  op _=_ : TypeList TypeList -> Equation [comm] .
  op _,_ : Equations Equations -> Equations [assoc comm id: empty] .
  var Eqn : Equation .  vars T T' T1 T1' : Type .  vars Tl Tl' : TypeList .
  eq Eqn, Eqn = Eqn .
  eq (T = T) = empty .
  eq (T,T1,Tl = T',T1',Tl') = (T = T'), (T1,Tl = T1',Tl') .
endfm
```

Defining the State Infrastructure

The state of the type inferencer will need to contain three attributes: a type environment binding names to generic types, a set of equational type constraints, and a counter giving the next "free" type variable. The state can be easily defined by extending the state of the type checker discussed in the previous section with the two additional attributes.

```
fmod TYPE-STATE is
  extending TYPE-ENVIRONMENT .
  extending EQUATIONS .
  sorts TypeStateAttribute TypeState .
  subsort TypeStateAttribute < TypeState .
  op empty : -> TypeState .
  op __ : TypeState TypeState -> TypeState [assoc comm id: empty] .
  op tenv : TypeEnv -> TypeStateAttribute .
  op eqns : Equations -> TypeStateAttribute .
  op nextType : Nat -> TypeStateAttribute .
endfm
```

Several standard helping operations are also needed and defined in a module `HELPING-OPERATIONS`. Only two helping operations are worthwhile discussing, namely:

```
  op freshTypeVar : Nat TypeState -> Pair .
  eq freshTypeVar(0, S) = {nil,S} .
 ceq freshTypeVar(s(#), nextType(N) S) = {(t(N),Tl), Sl}
  if {Tl,Sl} := freshTypeVar(#, nextType(N + 1) S) .
...
  op _&_ : TypeState Equations -> TypeState .
  eq (eqns(Eqns) S) & Eqns' = eqns(Eqns, Eqns') S .
```

`freshTypeVar` takes a natural number `#` and a state and generates `#` "fresh" types together with the new state (containing an updated counter for fresh types). This operation will be intensively used in the sequel. `S & Eqns` collects the the equations in `Eqns` into the state `S`.

Pre-Typing Generic Expressions

We next define the operation `preType` taking an expression and a state and returning a pair type-state, where the returned type is a temporary type calculated for the expression; that type will be "evaluated" into a concrete type only after solving the type constraints that are collected in the returned state.

The `preType` of integers and names is straightforward, because there is no constraint that needs to be collected:

```
eq preType(I, S) = {int, S} .
eq preType(X, S) = {S[X], S} .
```

As usual, we prefer to work with lists of expressions. Constraints must be properly accumulated when pre-typing lists of expressions:

```
ceq preType((E,E',El), S) = {(T,Tl), Sl}
 if {T,S'} := preType(E,S)
 /\ {Tl,Sl} := preType((E',El),S') .
```

Pre-Typing Arithmetic and Boolean Expressions

When we pre-type an arithmetic or boolean operator, there are two things that we should consider, both derived from the typing policy of the language:

1. What are the expected types of its arguments and of its result;
2. How to propagate the type constraints.

```
...
 ceq preType(E + E', S) = {int, S' & (T,T' = int,int)}
  if {(T,T'), S'} := preType((E,E'), S) .
...
 ceq preType(E  leq E', S) = {bool, S' & (T,T' = int,int)}
  if {(T,T'), S'} := preType((E,E'), S) .
...
```

Pre-Typing the Conditional

The following definition is self-explanatory: pre-type the condition to `Tb` and the two branches to `T` and `T'`, respectively, and then add the appropriate type constraints:

```
ceq preType(if BE then E else E', S) = {T, S' & (Tb,T = bool,T')}
 if {(Tb,T,T'), S'} := preType((BE,E,E'), S) .
```

Pre-Typing Functions

As usual, we have to consider the two language constructs related to functions, namely function declaration and function invocation. For function declarations, the returned pre-type is the function type calculated as follows:

```
ceq preType(fun Xl -> E, S) = {Tl -> Tr, Sr[tenv <- tenv(S)]}
 if {Tl,Sp} := freshTypeVar(#(Xl),S)
 /\ {Tr, Sr} := preType(E, Sp[Xl <- Tl]) .
```

Therefore, a "fresh" type is generated for the function's parameters, then the body of the function is pre-typed in the properly modified state, and then the resulting function type is returned. To be consistent with the semantics of `FUN`, the returned state *forgets* the parameter bindings. However, note that the type constraints are not forgotten! They will be needed at the end of pre-typing.

Function invocations are a bit tricky, because fresh types need to be generated. More precisely, a fresh type is generated for each function invocation, and it is assumed to be the type of the result. Knowing that type, we can generate the appropriate constraints:

```
ceq preType(F El, S) = {Tr, Sr & (Tf = Tl -> Tr)}
 if {(Tf,Tl), Sf} := preType((F,El), S)
 /\ {Tr,Sr} := freshTypeVar(1,Sf) .
```

It is important to understand the need for the fresh type generated for the result of the function. One may argue that it is not needed, because one can simply pre-type `F`, which should pre-type to a function type `Tp -> Tr`, then `E` to some `Tp'`, and then return the type `Tr` and generate the obvious constraint `Tp = Tp'`. However, the problem is that `E` *may not pre-type to a function type*!

Consider, for example, the expression `fun (x,y) -> (x y)`. When pre-typing `(x y)`, the only thing known about `x` is that it has some generic type, say `t(0)`. `y` has also a generic type, say `t(1)`. Then how about the type of `(x y)`? By generating a fresh type, say `t(2)`, for the result of the function application `(x y)`, we can conclude that the type of `x` should be `t(1) -> t(2)`, so we can generate the appropriate type constraint.

Pre-Typing Let

In order to pre-type `let`, we first pre-type each bound expression and collect all the generated type constraints, then we assign fresh generic types to the names to be bound and add new corresponding type constraints, and then finally pre-type the body of the `let`.

```
ceq preType(let(Xl,El,E), S) = {Te, Se[tenv <- tenv(S)]}
 if {Tel,Sel} := preType(El,S)
 /\ {Te,Se} := preType(E, Sel[Xl <- Tel]) .
```

Notice that there are no additional type constraints to be propagated, and that, like for functions, the state which is returned after pre-typing `let` *forgets* the bindings added to pre-type its body, but retains all the generated type constraints as well as the new value of the fresh type counter!

Pre-Typing Lists

As mentioned earlier, the typing policy of a list is that all its elements have the same type. To ensure this, we need to pre-type all the expressions in a list and to impose the constraints that all their types are equal. This can be done with one additional operator and a corresponding conditional equation:

```
 op preTypeL : ExpList TypeState -> [Pair] .
ceq preTypeL(E,El, S) = {T, S'' & (T = T')}
 if {T,S'} := preType(E, S)
 /\ {T',S''} := preTypeL(El, S') .
 eq preTypeL(nil,S) = freshTypeVar(1,S) .
```

The returned type can be any of the types that occurred in the list; we chose the first one. We have seen that the empty list should have a polymorphic type. We can ensure that easily by generating a "fresh" type for the pretype of an empty list.

Now a list can be pre-typed as follows:

```
ceq preType([El], S) = {list T, S'}
 if {T,S'} := preTypeL(El, S) .
```

The list operators can be pre-typed in a similar way. For example:

```
ceq preType(car(E), S) = {T?, S' & (list T? = T)}
 if {T?,S?} := freshTypeVar(1,S)
 /\ {T,S'} := preType(E, S?) .
```

Solving the Type Constraints

We can pre-type all the other language constructs in a very similar way (this will be part of your homework). Thus, the operator `preType` will take an expression and return a pair {`T,S`}, where `T` is a term of sort `Type` and `eqns(S)` contains all the *type constraints* accumulated by traversing the program and applying the typing policy. We refer to terms `t(0)`, `t(1)`, ..., as *type variables*.

We should understand the result {`T,S`} of `preType(E)` as follows:

> The (final) type of `E` will be `T` in which all the type variables will be *substituted* by the types obtained after *solving* the system of equations in `S`. `E` is *not guaranteed* a type *a priori*! If any *conflict* is detected while solving the system then we say that the process of typing `E` failed, and we conclude that the expression is not correctly typed.

If `Eqns` is a set of type equations, that is, a term of sort `Equations`, and `T` is a type potentially involving type variables, then we let `Eqns[T]` denote the type obtained after solving `Eqns` and then substituting the type variables accordingly in `T`.

Once such a magic operation `_[_] : Equations Type -> [Type]` is defined, we can easily find the desired type of an expression:

```
ceq type(E) = eqns(S)[T] if {T,S} := preType(E) .
```

We next describe a simple equational technique to define the operation `_[_] : Equations Type -> [Type]`.

Exercise 5 *Complete the definition of the type inferencer in* ***fun-type-inference-semantics.maude****. You need to pre-type the remaining language constructs and then to formalize the technique that will be next presented.*

The Type Inference Procedure by Examples

Example 1

Let us consider the following expression:

```
(fun (x,y) -> x y) (fun z -> 2 * z * z * 3, 3)
```

After pre-typing (do it, using `preType` on it), we get the pre-type `t(4)` and a state containing the three equations:

```
int = t(3),
t(0) = t(1) -> t(2),
(t(0),t(1)) -> t(2) = ((t(3) -> int),int) -> t(4)
```

How can we solve the system of equations above? We can immediately notice that `t(3) = int`, so we can just substitute `t(3)` accordingly in the other equations, obtaining:

```
t(0) = t(1) -> t(2),
(t(0),t(1)) -> t(2) = ((int -> int),int) -> t(4)
```

Moreover, since `t(0) = t(1) -> t(2)`, by substitution we obtain

```
((t(1) -> t(2)),t(1)) -> t(2) = ((int -> int),int) -> t(4)
```

Both types in the equation above are function types, so their source and target domains must be equal. This observation allows us to generate two other type equations, namely:

```
(t(1) -> t(2)),t(1) = (int -> int),int
t(2) = t(4).
```

Now, expanding the first equation into two equations and substituting `t(2)` by `t(4)`, we get:

```
t(1) -> t(4) = int -> int
t(1) = int
```

Substituting the second equation we get,

```
int -> t(4) = int -> int
```

that is, an equality of function types, which yields the following:

```
int = int
t(4) = int.
```

The first equation is useless and will be promptly removed by the simplifying rule `(T = T) = none` in `EQUATIONS`. The second equation gives the desired type of our expression, `int`.

Example 2

Let us now see an example expression which cannot be typed, e.g.:

```
let x = fun x -> x in (x x)
```

After pre-typing, we get the type `t(1)` constrained by:

```
t(0) -> t(0) = (t(0) -> t(0)) -> t(1)
```

We can already see that this equation is problematic, but let us just follow the procedure blindly to see what happens. Since both types in the equation are function types, their source and target types must be equal, so we can generate the following two equations:

```
t(0) = t(0) -> t(0),
t(0) = t(1)
```

The first equation is obviously problematic because of its "recursive" nature, but let us ignore it and substitute the second into the first, obtaining:

```
t(1) = t(1) -> t(1).
```

There is no way to continue. The variable `t(1)` cannot be assigned a type, so the original expression *cannot be typed*.

Unification

The technique that we used to solve the type equations is called *unification*. In general the equational unification problem can be stated as follows, where we only consider the mono-sorted case. Our particular unification problem fits this framework, because we can consider that we have only one sort, `Type`.

Let Σ be a signature over only one sort, i.e., a set of operations like in Maude, let X be a finite set of variables, and let

$$\mathcal{E} = \{t_1 = t'_1, ..., t_n = t'_n \mid t_1, t'_1, ..., t_n, t'_n \in T_\Sigma(X)\}$$

be a finite set of pairs of Σ-terms over variables in X, which from now on we call *equations*. Notice, however, that these are different from the standard equations in equational logic, which are quantified universally.

A map, or a substitution, $\theta : X \to T_\Sigma(X)$ is called a *unifier* of $\mathcal{E}$ if and only if $\theta^\star(t_i) = \theta^\star(t'_i)$ for all $1 \leq i \leq n$, where $\theta^\star : T_\Sigma(X) \to T_\Sigma(X)$ is the natural extension of $\theta : X \to T_\Sigma(X)$ to entire terms, by substituting each variable x by its corresponding term, $\theta(x)$.

A unifier $\theta : X \to T_\Sigma(X)$ is *more general* than $\varphi : X \to T_\Sigma(X)$ when φ can be obtained from θ by further substitutions; formally, when there is some other map, say $\rho : X \to T_\Sigma(X)$, such that $\varphi = \theta; \rho^\star$, where the composition of function was written sequentially. Let us consider again our special case signature, that of types, and the equations $\mathcal{E}$:

```
t(0) = t(1) -> t(2),
t(0) -> t(1) -> t(2) = (t(3) -> t(3)) -> (t(4) -> t(4)) -> t(5).
```

Here `t(0)`, ..., `t(5)` are seen as variables in X.

One unifier for these equations, say φ, takes `t(0)` to

```
(int -> int) -> (int -> int),
```

`t(1)`, `t(2)` and `t(3)` to `int -> int`, and `t(4)` to `int`. Another unifier, say θ, takes `t(0)` to

```
(t(4) -> t(4)) -> (t(4) -> t(4)),
```

and `t(1)`, `t(2)` and `t(3)` to `t(4) -> t(4)`. Then it is clear that θ is *more general* than φ, because $\varphi = \theta; \rho^\star$, where ρ simply takes `t(4)` to `int`.

If a set of equations $\mathcal{E}$ admits a unifier, they are called *unifiable*. Note that there can be situations, like the ones we have already seen for our special case of signature of types, when a set of equations is not unifiable.

Finding the Most General Unifier

When a set of equations $\mathcal{E}$ is unifiable, its *most general unifier*, written $mgu(\mathcal{E})$ is one which is more general than any other unifier of $\mathcal{E}$. Note that typically there are more than one *mgu*.

We will next discuss a general procedure for finding an *mgu* for a set of equations $\mathcal{E}$ over an arbitrary mono-sorted signature Σ. Since we actually need to apply that *mgu* to the type calculated by the `preType` operation for the expression to type, we will provide a technique that directly calculates $\mathcal{E}[t]$ for any term t, which calculates the term after applying the substitution $mgu(\mathcal{E})$ to t, that is, $(mgu(\mathcal{E}))^\star(t)$.

The technique is in fact quite simple. It can be defined entirely equationally, but in order to make the desired direction clear we write them as rewriting rules (going from left to right). It consists of iteratively applying the following two steps to $\mathcal{E}[t]$:

1. $(x = u, \mathcal{E})[t] \rightarrow subst(x, u, \mathcal{E})[subst(x, u, t)]$, if x is some variable in X, u is a term in $T_\Sigma(X)$ which does *not* contain any occurrence of x, and $subst(x, u, \mathcal{E})$ and $subst(x, u, t)$ substitute each occurrence of x in $\mathcal{E}$ and t, respectively, by u;

2. $(\sigma(t_1, ..., t_k) = \sigma(t'_1, ..., t'_k), \mathcal{E})[t] \rightarrow (t_1 = t'_1, ..., t_k = t'_k, \mathcal{E})[t]$, if $\sigma \in \Sigma$ is some operation of k arguments.

Theorem. *When none of the steps above can be applied anymore, we obtain a potentially modified final set of equations and a final term, say $\mathcal{E}_f[t_f]$. If $\mathcal{E}_f$ is empty then t_f is $(mgu(\mathcal{E}))^\star(t)$, so it can be returned as the type of the original expression. If $\mathcal{E}_f$ is not empty then $\mathcal{E}$ is not unifiable, so the original expression cannot be typed.*

Let Polymorphism

Unfortunately, our current polymorphic type inference technique is not as general as it could be. Suppose, for example, that one wants to define a function with the purpose of using it on expressions of various types, such as an identity function. Since the identity function types to a polymorphic type, `t(0) -> t(0)` or some similar type, one would naturally want to use it in different contexts. After all, that's the purpose of polymorphism. Unfortunately, our current type inferencer *cannot* type the following expression, even though it correctly evaluates to `int(1)`:

```
let f = fun x -> x
in if f true then f 1 else f 2
```

The reason is that `f true` will already enforce, via the type constraints, the type of `f` to be `true -> true`; then `f 1` and `f 2` will enforce the type of `f` to be `int -> int`, thus leading to a contradiction. Indeed, our type inferencer will reduce $\mathcal{E}[t]$, where $\mathcal{E}$ is the set of constraints and t is the pre-type of this expression, to $\mathcal{E}_f[t_f]$, where $\mathcal{E}_f$ is the equation `bool=int` and t_f is `int`; since $\mathcal{E}_f$ is not empty, the original expression cannot be typed.

The example above may look artificial, because one does not have a need to define an identity function. Consider a length function instead, which is, naturally, intended to work on lists of any type:

```
fun x -> let c = 0
         in {
              while(not null?(x))
              {
                x := cdr(x) ;
                c := c + 1
              } ;
              c
            }
```

As expected, our type inferencer will type this expression to a polymorphic type, `list t(0) -> int`. However, if one uses it on a list of concrete type, then `t(0)` will be unified with that type, thus preventing one from using the length function on other lists of different types. For example,

```
let l = fun x -> let c = 0
                 in {
                      while(not null?(x))
                      {
                        x := cdr(x) ;
                        c := c + 1
                      } ;
                      c
                    }
in l [5 :: 3 :: 1] + l [true :: false :: true]
```

will not type, even though it correctly evaluates to `int(6)`.

This leads to the following natural question, that a programming language designer wishing a type system on top of his/her language must address: what should one do when valid programs are rejected by the type system? One unsatisfactory possibility is, of course, to require the programmers write their programs differently, so that they will be accepted by the type system. Another, better possibility is to improve the type system to accept a larger class of programs.

Since *reusing* is the main motivation for polymorphism, it would practically make no sense to disallow the use of polymorphic expressions in different contexts. Therefore, programming language scientists fixed the problem above by introducing the important notion of *let-polymorphism*.

The idea is to type the `let` language constructs differently. Currently, we first type all the binding expressions to some (pre-)types, then we bind those types to the corresponding names into the type environment, and finally we type the body expression in the new environment. This technique is inherently problematic with respect to reuse, because a (polymorphic) binding expression is enforced to have two different types if used in two different contexts, which is not possible.

The idea of *let-polymorphism* is to first *substitute* all the free occurrences of the bound names into the body of the `let` expression by the corresponding binding expressions, and then to type directly the body expression. This way, each use of a bound name will be typed locally, thus solving our problem.

Exercise 6 *Modify our current type inferencer to include let-polymorphism. How about* `letrec`*?*

However, a complexity problem is introduced this way: the size of the code can grow exponentially and the binding expressions will be (pre-)typed over and over again, even though each time they will be typed to "almost" the same type (their types will differ by only the names of the type variables). There is a partial solution to this problem, too, but that is a more advanced topic that will be covered in CS522 next semester.

Type Systems May Reject Correct Programs

While static type checkers/inferencers offer a great help to programmers to catch (trivial) errors at early stages in their programs and to programming language implementers to generate more efficient code (by assuming the untyped memory model), they have an inherent drawback: they *disallow correct programs which do not type check*. In the previous section we saw a fortunate case where a type system could be relatively easily extended (let-polymorphism), to allow polymorphic definitions to be used as desired. Unfortunately, due to undecidability reasons, for any given (strong) type system, there will always be programs which can evaluate correctly but which will not type check.

In our context, for example, consider the following expression/program for calculating the length of a list *without* using `letrec` or loops:

```
let t f x = if null?(x) then 0 else 1 + f f cdr(x)
in let l x = t t x
   in l [4 :: 7 :: 2 :: 1 :: 9]
```

The idea in the program/expression above is to pass the function as an argument to itself; this way, it will be available, or "seen", in its own body, thus simulating the behavior of `letrec` without using it explicitly. That is how (functional) programmers wrote recursive programs in the early days of functional programming, when the importance of static scoping was relatively accepted but no clean support for recursion was available. It, indeed, evaluates to `int(5)` but does not type. Try it!

One may wrongly thing that the above does not type because our type system does not include let-polymorphism. To see that let-polymorphism is not the problem here, one can apply the substitutions by hand and thus eliminate the `let` constructs entirely:

```
(fun f x -> if null?(x) then 0 else 1 + f f cdr(x))
  (fun f x -> if null?(x) then 0 else 1 + f f cdr(x))
    [4 :: 7 :: 2 :: 1 :: 9]
```

The above will again evaluate to `int(5)`, but will not type. The set of constraints is reduced to `t(6) = t(6) -> list int -> int`, reflecting the fact that there is some expression whose type has a recursive behavior. Indeed, the type of `f` (any of them) cannot be inferred due to its recursive behavior.

In fact, the problematic expression is

```
fun f x -> if null?(x) then 0 else 1 + f f cdr(x)
```

which cannot be typed because of the circularity in the type of `f`. No typed functional languages can actually type this expression. In ML, for example, the same expression, which in ML syntax is

```
fn (f,x) => if null(x) then 0 else 1 + f f tl(x)
```

generates an error message of the form:

```
t.sml:16.36-16.49 Error: operator is not a function [circularity]
  operator: 'Z
  in expression:
    f f
```

Modifying the K Language Definition Into a Type Inferencer

Figure 12 shows a translation of the K language definition from Figure 10 and/or of the K definition of the type checker in Figure 11 into a K-definition of a type inferencer.

Exercise 7 *Translate the K-definition of the type inferencer in Figure 12 into Maude.*

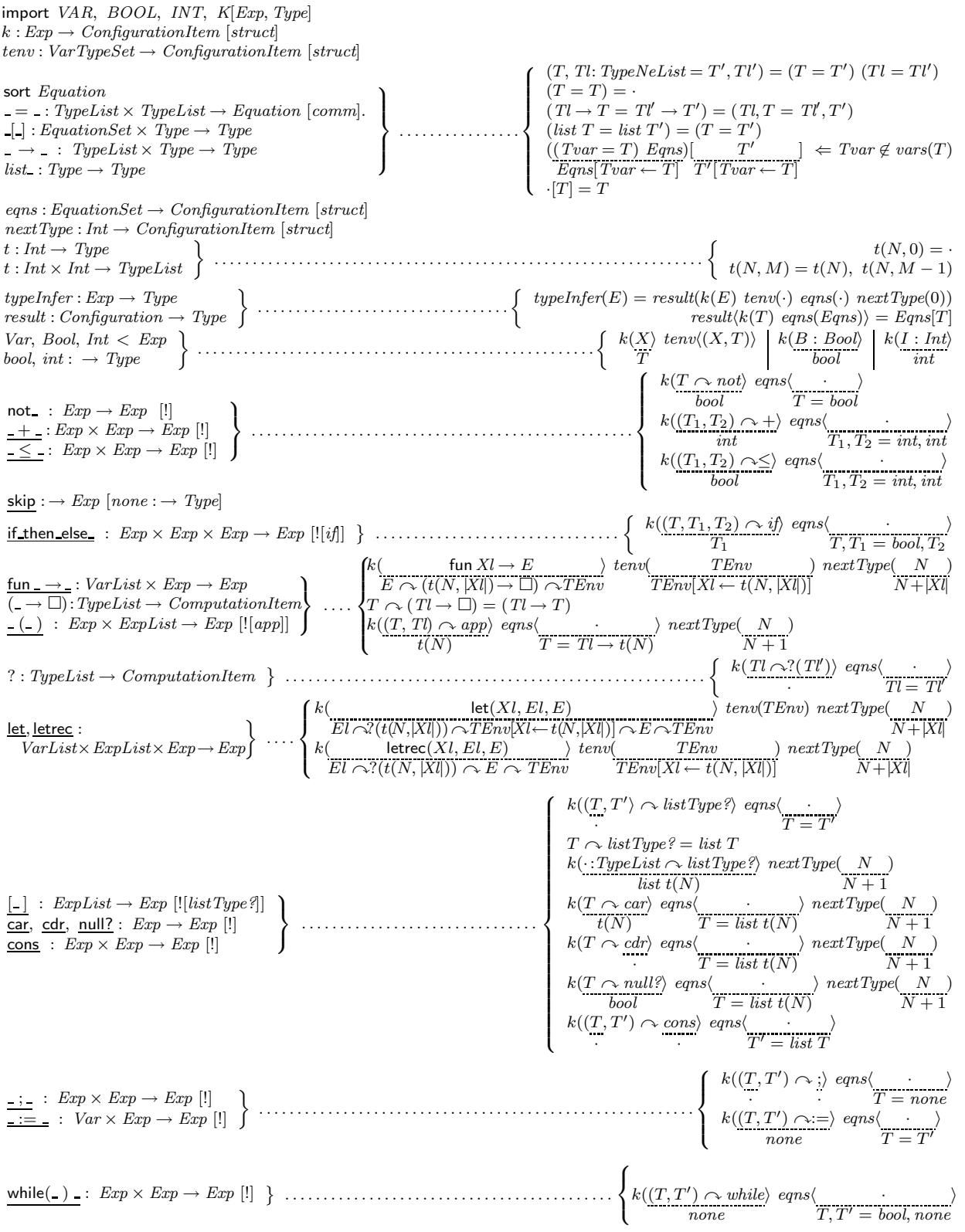

Figure 12: K definition of a type inferencer for the simple functional language